MetadataConfiguration

File(s): *conf/metadata-providers.xml*

Format: Custom Schema

Overview

The Shibboleth IdP generally requires SAML metadata to provision connectivity with relying parties and inform it about their capabilities and technical specifics. While you have the option to operate in a more "promiscuous" way (by enabling profiles for "unverified" RPs), this is relatively rare. In most cases, you will configure metadata sources in order to use the IdP's SAML features; this is done by adding `<MetadataProvider>` elements inside the *metadata-providers.xml* file.

In a typical (and default) configuration, only one resource is configured for metadata configuration, a file called *metadata-providers.xml* containing a "root" `<MetadataProvider>` element. This could make use of a single resource such as a local file:

Simple file based configuration

```
<MetadataProvider id="LocalMetadata" xsi:type="FilesystemMetadataProvider"
 xmlns="urn:mace:shibboleth:2.0:metadata"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="urn:mace:shibboleth:2.0:metadata
http://shibboleth.net/schema/idp/shibboleth-metadata.xsd"
 failFastInitialization="true"
 metadataFile="%{idp.home}/metadata/my-metadata.xml"/>
```

It could also point to a third-party-provided metadata source, usually referred to as a "federation", with filters to verify the signature on the XML document using a pre-established public key:

Loading metadata from a federation metadata server

```
<MetadataProvider id="FederationMetadata"
xsi:type="FileBackedHTTPMetadataProvider"
 xmlns="urn:mace:shibboleth:2.0:metadata"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="urn:mace:shibboleth:2.0:metadata
http://shibboleth.net/schema/idp/shibboleth-metadata.xsd"
 metadataURL="http://federation.example.org/metadata.xml"
 backingFile="%{idp.home}/metadata/federation-metadata.xml">

    <MetadataFilter xsi:type="SignatureValidation" requireSignedRoot="true"
  certificateFile="%{idp.home}/conf/metadata/fed-signing-key.pem"/>

    <MetadataFilter xsi:type="RequiredValidUntil"
maxValidityInterval="P14D"/>
</MetadataProvider>
```

One can combine the two approaches inside a ChainingMetadataProvider, which combines two or more sources.

Reference

Schema names and locations

Nearly all elements described in this page and its children are defined in the `urn:mace:shibboleth:2.0:metadata` namespace, the schema for which can be located at http://shibboleth.net/schema/idp/shibboleth-metadata.xsd. Throughout this document and its children, this is assumed to be the default XML namespace in effect. The namespace prefix `"metadata:"` is conventionally also bound to this namespace.

The namespace prefix "`security:`" is used to refer to the `urn:mace:shibboleth:2.0:security` namespace, the schema for which can be located at http://shibboleth.net/schema/idp/shibboleth-security.xsd, and is generally used only in advanced scenarios or for compatibility.

The namespace prefix "`resource:`" is used to refer to the `urn:mace:shibboleth:2.0:resource` namespace, the schema for which can be located at http://shibboleth.net/schema/idp/shibboleth-resource.xsd. It is only used by the `ResourceBackedMetadataProvider`

The namespace prefix "`samlmd:`" is used to refer to the `urn:oasis:names:tc:SAML:2.0:metadata` namespace, the schema for which can be located at http://docs.oasis-open.org/security/saml/v2.0/saml-schema-metadata-2.0.xsd

MetadataProvider Plugin Types

The precise behavior of any `<MetadataProvider>` element is controlled by the `xsi:type` attribute (see below). The following types are supported and examples are provided for each type. If the `urn:mace:shibboleth:2.0:metadata` namespace is not the default, then a prefix (presumably "metadata:") is required when specifying these types.

xsi:type	Function
ChainingMetadataProvider	Allows multiple sources to be specified, generally the top-level type used
DynamicHTTPMetadataProvider	Dynamically fetches metadata from a suitably configured HTTP server acting as a metadata query service
LocalDynamicMetadataProvider	Dynamically fetches metadata from a local source, such as a filesystem directory, populated and maintained by the IdP deployer
FilesystemMetadataProvider	Loads (and reloads) metadata from the filesystem
HTTPMetadataProvider	Periodically pulls metadata from a given URL
FileBackedHTTPMetadataProvider	Extends the HTTPMetadataProvider by saving the metadata locally where it can be reloaded if the source is unavailable

InlineMetadataProvider	Allows metadata to be specified inline
ResourceBackedMetadataProvider	Allow access to metadata from SVN and other complex sources

Attributes

The following attributes available on all metadata provider types:

Name	Type	Default	Description
id	String		Identifier for the metadata source for logging, identification for command line reload, etc.
xsi:type	String		Specifies the exact type of provider plugin to use (from those listed above, or a custom extension type).
requireValidMetadata	Boolean	true	Whether candidate metadata found by the resolver must be valid in order to be returned (where validity is implementation specific, but in SAML cases generally depends on a `validUntil` attribute .) If this flag is true, then invalid candidate metadata will not be returned.
failFastInitialization	Boolean	true	Whether to fail initialization of the underlying MetadataResolverService (and possibly the IdP as a whole) if the initialization of a metadata provider fails. When false, the IdP may start, and will continue to attempt to reload valid metadata if configured to do so, but operations that require valid metadata will fail until it does.
sortKey	Integer		Defines the order in which metadata providers are searched (see below), can only be specified on top level `<MetadataProvider>` elements.
The following are advanced settings supporting a new low-level feature allowing metadata lookup by keys other than the unique entityID and are rarely of use to a deployer.			
criterionPredicateRegistryRef [3.3]	Bean ID		Identifies the a custom `CriterionPredicateRegistry` bean used in resolving predicates from non-predicate input criteria.
useDefaultPredicateRegistry [3.3]	Boolean	true	Flag which determines whether the default `CriterionPredicateRegistry` will be used if a custom one is not supplied explicitly.
satisfyAnyPredicates [3.3]	Boolean	false	Flag which determines whether predicates used in filtering are connected by a logical 'OR' (true) or by logical 'AND' (false).
The following attributes are supported only on reloading "batch-oriented" metadata providers (ResourceBackedMetadataProvider, FilesystemMetadataProvider, and both HTTPMetadataProviders, including the FileBackedHTTPMetadataProvider):			
Name	**Type**	**Default**	**Description**
parserPoolRef	Bean ID	shibboleth.ParserPool	Identifies a Spring bean for the (OpenSAML) `ParserPool` object used to parse incoming metadata. Generally should not be changed.
taskTimerRef	Bean ID		Identifies a Spring bean containing a Java `TaskTimer` used to schedule reloads. When not set, an internal timer is created. Generally should not be changed.
refreshDelayFactor	Real Number (strictly between 0.0 and 1.0)	0.75	A factor applied to the initially determined refresh time in order to determine the next refresh time (typically to ensure refresh takes place prior to the metadata's expiration). Attempts to refresh metadata will generally begin around the product of this number and the maximum refresh delay.
maxRefreshDelay	Duration	PT4H	Upper bound on the next refresh from the time calculated based on the metadata's expiration.
minRefreshDelay	Duration	PT30S	Lower bound on the next refresh from the time calculated based on the metadata's expiration.
indexesRef [3.3]	Bean ID		Identifies an optional `Set<MetadataIndex>` used to support resolution of metadata based on criteria other than an entityID.
resolveViaPredicatesOnly [3.3]	Boolean	false	Flag indicating whether resolution may be performed solely by applying predicates to the entire metadata collection, when an entityID input criterion is not supplied.

expirationWarningThreshold [3.4]	Duration	PT0S (disabled)	For each attempted metadata refresh (whether or not fresh metadata is obtained), if `requireValidMetadata` is true, and there is a `validUntil` XML attribute on the document root element, and the difference between `validUntil` and the current time is less than `expirationWarningThreshold`, the system logs a warning about the impending expiration.
The following attributes are supported only on the dynamic metadata providers (DynamicHTTPMetadataProvider and LocalDynamicMetadataProvider):			
Name	**Type**	**Default**	**Description**
parserPoolRef	Bean ID	shibboleth.ParserPool	Identifies a Spring bean for the (OpenSAML) `ParserPool` object used to parse incoming metadata. Generally should not be changed.
taskTimerRef	Bean ID		Identifies a Spring bean containing a Java `TaskTimer` used to schedule reloads. When not set, an internal timer is created. Generally should not be changed.
refreshDelayFactor	Real Number (strictly between 0.0 and 1.0)	0.75	A factor applied to the initially determined refresh time in order to determine the next refresh time (typically to ensure refresh takes place prior to the metadata's expiration). Attempts to refresh metadata will generally begin around the product of this number and the maximum refresh delay.
minCacheDuration	Duration	PT10M (10 minutes)	The minimum duration for which metadata will be cached before it is refreshed.
maxCacheDuration	Duration	PT8H (8 hours)	The maximum duration for which metadata will be cached before it is refreshed.
maxIdleEntityData	Duration	PT8H (8 hours)	The maximum duration for which metadata will be allowed to be idle (no requests for it) before it is removed from the cache.
removeIdleEntityData	Boolean	true	Flag indicating whether idle metadata should be removed.
cleanupTaskInterval	Duration	PT30M (30 minutes)	The interval at which the internal cleanup task should run. This task performs background maintenance tasks, such as the removal of expired and idle metadata.
persistentCacheManagerRef [3.3]	Bean ID		The optional manager for the persistent cache store for resolved metadata. On metadata provider initialization, data present in the persistent cache will be loaded to memory, effectively restoring the state of the provider as closely as possible to that which existed before the previous shutdown. Each individual cache entry will only be loaded if 1) the entry is still valid as determined by the internal provider logic, and 2) the entry passes the (optional) predicate supplied via `initializationFromCachePredicateRef`.
persistentCacheManagerDirectory [3.3]	File specification		The directory used for an internally-constructed filesystem-based persistent cache. This is a convenience parameter to avoid specifying a full bean via `persistentCacheManagerRef`. This option will be ignored if `persistentCacheManagerRef` is specified.
persistentCacheKeyGeneratorRef [3.3]	Bean ID	internal default instance	Identifies a Spring bean for a `Function` which generates the string key used with the cache manager. The default implementation produces the lower-case hex-encoded SHA-1 digest of the entityID of the `EntityDescriptor`.
initializeFromPersistentCacheInBackground [3.3]	Boolean	true	Flag indicating whether should initialize from the persistent cache in the background. Initializing from the cache in the background will improve IdP startup times.
backgroundInitializationFromCacheDelay [3.3]	Duration	PT2S (2 seconds)	The delay after which to schedule the background initialization from the persistent cache when `initializeFromPersistentCacheInBackground=true`.
initializationFromCachePredicateRef [3.3]	Bean ID	an "always true" predicate	Identifies a Spring bean for an optional `Predicate` which determines whether a given entity should be loaded from the persistent cache at resolver initialization time.

Other attributes are specific to the `xsi:type`, and these are documented on the pages specific to each type.

Child Elements

The following child elements can be used with all metadata provider types:

Name	Cardinality	Description
<MetadataFilter>	any	Metadata filter plugins to run
<security:TrustEngine>	any	Trust engine plugins to use in a separate SignatureValidation filter's trustEngineRef attribute

Other allowable child elements are specific to the `xsi:type` of the MetadataProvider used, and these are documented on the pages specific to each type.

Miscellany

Multiple Configuration Files

As described in the ReloadableServices documentation, the configuration is actually loaded from a bean whose name is specified by the property **idp.service.metadata.resources**, with the default value **shibboleth.MetadataResolverResources** which in turn is defined in services.xml to be a list with one entry: the file *metadata-providers.xml*

You can, if you choose, override this with additional or different files or more advanced sources. Each resource must supply a "top level" `<MetadataProvider>` element with attributes and child elements as described above. Search order amongst multiple top level elements is arbitrated by the `sortKey` attribute, where lower values are processed before higher ones.

Search Ordering

If a specific relying party (as identified by a specific entityID) is duplicated in the metadata sources provided, then which precise entry is chosen is governed by the following rules:

- Metadata sources combined via a chain are searched in the order in which they occur in the chain, and the first entry matching the entityID is returned.
- If multiple "top level" Metadata Providers are provided then they are searched in an order derived from the (numeric) value of the `sortKey` attribute (lowest key first). If no `sortKey` is specified, then the search order is undefined.
- In whatever order of sources is in effect, the first entry matching the entityID is returned.
- If a single metadata source contains multiple entries with the same entityID, then which entry is returned is undefined.

V2 Compatibility

A single `<MetadataProvider>` element may be embedded in a legacy *relying-party.xml* file as described in the older documentation. Consult the V2 documentation for this, and do not mix and match this approach with newer configuration features.

During the V2 to V3 upgrade process, the original V2 *relying-party.xml* file is copied to *metadata-providers.xml*, to serve as the metadata configuration for the new version. It's strongly advisable after upgrading to update that file by stripping it of the older content and promote the `<MetadataProvider>` element in it to the root of the file. In the interim all other content in the file except for `<MetadataProvider>` elements (and any referenced `<security:TrustEngine>` elements) is ignored.

The following non-relevant trust engine types often found in a legacy relying-party.xml file are ignored if seen and cannot be used for metadata verification:

- `Chaining`
- `MetadataExplicitKey`
- `MetadataPKIXX509Credential`
- `MetadataExplicitKeySignature`
- `MetadataPKIXSignature`
- `StaticPKIXX509Credential`

New Capabilities in V3

The V3 metadata configuration syntax is backward-compatible with the V2 `<MetadataProvider>` syntax and adds some useful new shortcuts.

You can provide multiple metadata configuration files (not just multiple metadata sources in one file), as described above.

When configuring more than one MetadataFilter, you need not wrap them in a "ChainingFilter" filter.

The SignatureValidation filter need not contain a `trustEngineRef` attribute referencing a separately-defined trust engine, instead a certificate file may be specified directly with the `certificateFile` attribute, or a PEM-format public key may be supplied inline via the `<PublicKey>` element. Finally, for more advanced requirements, a `<security:TrustEngine>` element that is today referenced from a `<MetadataProvid`

er> but declared outside of it may be a child element of the <MetadataProvider> element.

Notes

TBD